

DUKE NUKEM 2 TOTALLY UNOFFICIAL GAME FILE SPECS

=====

Compiled Sept. 2002 based on hacking and research and notes done years ago. Maybe someone will take up the torch and finally write the editor that I had planned to do, but never did. If you don't understand the info contained in this file then just delete it - it's not meant for you. Those who need it will understand it. 'Nuff said.

Have fun,

Dave Bollinger

DISCLAIMER:

=====

The information presented here is totally unofficial and without warranty of any kind and has been shown to cause cancer in laboratory animals. Any damage you do to your game files is entirely your own fault.

CONTACT:

=====

Feel free to email me with technical questions at:

davebollinger@hotmail.com

and I'll answer as best I can, if I can, if I feel like it. But please don't spam me or otherwise abuse that email account.

COMPACTED FILE SPECS:

=====

Many of Apogee's "classic" games use the same compacted file format with a bunch of individual files all crammed together into a single big file. Each of these compacted files begins with a directory of the individual files:

```
typedef struct {
    char filename[12];    // 12 bytes
    long fileoffset;      // 4 bytes
    long filesize;        // 4 bytes
} TDirectoryEntry;      // 20 bytes
```

```
#define NUM_DIRECTORY_ENTRIES 200
```

```
typedef TDirectoryEntry TDirectory[NUM_DIRECTORY_ENTRIES]; // 4000 bytes
```

```
TDirectory Directory;
```

```
// note: the file entry just after the last used entry contains
// the number of files in the pack stored as ascii in the filename.
// so, best way to check for last entry is 0 fileoffset and filesize.
// what a wacky way to do it, too bad the number of entries used
// isn't just an int at the top of the directory so you'd know
// ahead of time how many entries exist without having to scan.
```

```
// example usage:
```

```
in = fopen("nukem2.cmp", "rb");
```

```

fread(&Directory, sizeof(TDirectory), 1, in);
for (i=0; i<NUM_DIRECTORY_ENTRIES; i++)
{
    if (!fileoffset || !filesize)
        break;
    printf("%12.12s", Directory[i].filename);
}

```

(full source for the unpacker i used is below somewhere)

NOTE ON UNPACKED FILES:

=====

Most, if not all, of the games will check for the presence of the individual file before reading it from the compacted library. This is a way cool feature for hackers! Unpack the files, then hack away at the individual files, then start the game and it'll read them. (So go hack Cosmo too, it's not that different from DN2, except that it only supports a single global tileset, no tileset-per-level as with dn2's czones)

LEVEL DATA SPECS:

=====

Level files are named like:

```

L#.MNI for episode 1
M#.MNI for episode 2
N#.MNI for episode 3
O#.MNI for episode 4

```

and are around 65-68K in size.

Each level starts with a level header of 47 bytes:

```

typedef struct
{
    WORD dataoffset;    // offset within this file to levelwidth & start of level
    data
    char graphics[13];  // name of czone tileset used for this level
    char backdrop[13];  // name of backdrop image used for this level
    char music[13];     // name of music file used for this level
    char unknown[4];    // unknown
    WORD actorwords;    // number of words of actor data (UNRELIABLE)
} TLevelHeader;

```

The header is followed by a variable number of actor headers of 6 bytes each:

```

typedef struct
{
    WORD objectid;      // object identifier - see codes listed below
    WORD xpos;          // initial x position of actor in level in tile units
    WORD ypos;          // initial y position of actor in level in tile units
} TActorHeader;

```

The level header's dataoffset element points to the byte following the last actor header record and can be used to calculate the number of actor records:

```

int actordatabytes = levelheader.dataoffset - sizeof(TLevelHeader);
int numactorrecords = actordatabytes / sizeof(TActorHeader);

```

UPDATED

The actorwords field of the level header is not reliable, in particular the original level N7.MNI doesn't add up correctly, so it appears better to instead calculate the number of actor bytes and number of actors as shown.

END UPDATE

The actors are stored in order based on their coordinates, starting with the top-left-most actor and running top-to-bottom, left-to-right, and ending with the bottom-right-most actor. I believe this is important because some "actors" modify other actors that they are adjacent to, like difficulty level modifiers, but may just be a result of the editor used storing the actors in a grid instead of a list and just dumping them out in row-col order.

The first word at the dataoffset position is the level width in tiles. It will be a power of 2 such as 64,128,256,512,1024. (level height will be explained in a moment - it is a derived value)

Following the level width is the actual level data. Each cell of the level is represented by a WORD. The entire level is 32750 words in size. (or 65500 bytes):

```
#define LEVEL_DATA_BYTES 65500
#define LEVEL_DATA_WORDS 32750
```

```
typedef struct
{
    WORD data[LEVEL_DATA_WORDS];
} TWorldData;
```

Note that the level is NOT an even 64K in length. The height of the level is calculated by dividing the size of the level data by the level width:

```
int levelheight = (int)(LEVEL_DATA_WORDS / levelwidth);
```

The unused end portion of the level data is usually zero-padded, but not always, it doesn't appear to matter. (for instance, if the level is 256x127 in dimension, it will only use 32512 words, the remaining 238 words (32750-32512) are unused.

That ends the "useful" portion of the level. In fact, if you chop off the level file at this point the game still appears to run just fine. (try it) But there's some more junk that occurs afterwards, it may just be stuff left over from the original editor or have some more obscure as-yet-unknown purpose within the game.

After the level data is a mysterious variable-length section. This section ranges from a couple hundred bytes to over 1400 bytes. The purpose of this section is unknown to me, other than it definately doesn't look like level data.

UPDATED

This variable data section begins with a WORD containing the length of the remaining variable portion. (in other words, 2 bytes to store the length + length bytes of variable data = total size of the post-level-data variable portion) Until someone figures out this variable section, I'd suggest a good way to "fake" it is write out 0 as the length, then append the CZone structure.

END UPDATE

The level file ends with info about the contents of it's czone file:

```
typedef struct
{
    char attr_filename[13];    // fe: zone1atr.mni
    char tile_filename[13];    // fe: zone1.mni
    char mask_filename[13];    // fe: zone1msk.mni
} TCZoneContents;            // 39 bytes
```

Why the contents of the czone file are stored in the level file is beyond me. It's redundant, because several levels use the same czone file. It's also completely unnecessary as the format is the same for all czone files.

/* end of level file */

YOUR FIRST EDIT:
=====

Open L1.MNI into your favorite hex editor and go to address 0x74F. This is the low-order byte of the object id for actor 304. This is the crate with the bomb that appears just to the right of Duke at the start of the level. Change this byte from the value 0x2A to 0x14 and you'll turn that bomb into a flamethrower. Save the file, go play it, and prove it to yourself.

GRAPHICS COMMENTS:
=====

DN2 requires a VGA card, but the game engine itself is still essentially a 16-color EGA graphics engine with custom palettes courtesy of the VGA adapter. The screens that you might have thought were 256-color are really just very cleverly crafted 16 color screens. The upside is, if you've hacked either Duke1 or Cosmo or Bash or Agent, or any other early Apogee EGA game, then the graphics format used here should be completely familiar.

PALETTE SPECS:
=====

DN2 uses a custom 16-color palette. It can be found in GAMEPAL.PAL (48 bytes, 16 RGB values)
This palette is used through most of the game except for fullscreen "splash" images as described below. Simply set the first 16 RGB DAC's to these values and the images will appear correctly. The tiles and screens do NOT look correct without the appropriate custom palette.

CZONE SPECS:
=====

CZone files are compacted tilesets. Each CZone file contains 3 things: the tile attributes, the non-masked solid tiles, and the masked transparent tiles:

```
typedef struct
```

```

{
  BYTE attribute_data[3600];      // exact format unknown, bitmasks indicating
solidity of tiles and such
  BYTE solid_tile_data[32000];    // 25 rows of 40 tiles, 4 planes, 1 byte wide,
8 bytes tall
  BYTE mask_tile_data[6400];     // 4 rows of 40 tiles, 5 planes, 1 byte wide,
8 bytes tall
} TCZoneContents;                // 42000 bytes

```

All tiles are 8x8 pixels, 1x8 bytes, stored in planar order with either 4, or 5 if masked, planes.

Here's some QBASIC code, originally written by...

Frenkel Smeijers
frenkel_smeijers@hotmail.com
<http://www.student.tue.nl/u/a.f.smeijers/sfp>

...to view Cosmo's pictures - now modified to view CZONE files. At one point I had C source to do this, but have since lost it, so it was easier just to borrow Frenkel's code. (thanks) I'm not much of a QB programmer, but I think it gets the job done, and at least demos the reading of this file format, setting the palette, etc, maybe Frenkel can clean it up.

(note that the use of screen 12 (VGA 640x480) is intentional, even when the output is only 320x200. need a VGA mode to set the palette, screen 7 (EGA 320x200) won't work. does QB support the 320x200 planar 16-color VGA mode?)

```

' VIEW_CZN.BAS
SCREEN 12

' PALETTE
DIM palet AS STRING * 1
OPEN "gamepal.pal" FOR BINARY AS #1
OUT &H3C8, 0
FOR i% = 1 TO 48
  GET #1, , palet
  OUT &H3C9, ASC(palet)
NEXT i%
CLOSE #1

OPEN "czone1.mni" FOR BINARY AS #1
DIM value AS STRING * 1
DEF SEG = &HA000

' ATTRIBUTES - SKIP
FOR skip% = 1 TO 3600
  GET #1, , value
NEXT skip%

' SOLID TILES
FOR row% = 0 TO 24
  FOR col% = 0 TO 39
    FOR y% = 0 TO 7
      FOR plane% = 0 TO 3
        OUT &H3C4, 2
        OUT &H3C5, INT(2 ^ plane%)*INT(2 ^ plane% - (2 ^ (plane% - 1)))
        GET #1, , value
        POKE (row% * 8 * 80) + (col%) + (y% * 80), ASC(value)
      NEXT plane%
    NEXT y%
  NEXT col%
NEXT row%

```

```

NEXT row%

' MASKED TILES
lastrow% = row%
FOR row% = lastrow% TO lastrow% + 16
  FOR col% = 0 TO 39
    FOR y% = 0 TO 7
      FOR plane% = 0 TO 4
        OUT &H3C4, 2
        OUT &H3C5, INT(2 ^ plane% - (2 ^ (plane% - 1)))
        GET #1, , value
        POKE (row% * 8 * 80) + (col%) + (y% * 80), ASC(value)
      NEXT plane%
    NEXT y%
  NEXT col%
NEXT row%

' WAIT
DO: LOOP WHILE INKEY$ = ""
DEF SEG
CLOSE #1
END ' OF VIEW_CZN.BAS

```

FULLSCREEN IMAGE SPECS:
=====

The fullscreen images (intro/help/titles/bonus/etc) are much like the screens from Duke1 or Cosmo. Stored in planar priority, with 4 planes, 8000 bytes per plane, (320x200), followed by a 48-byte palette specific to this image. Identify these files by filesize = 32048 bytes.

```

' VIEW_SCR.BAS
SCREEN 12
OPEN "weapons1.mni" FOR BINARY AS #1
DIM value AS STRING * 1
DEF SEG = &HA000

' PIXEL DATA
FOR plane% = 0 TO 3
  OUT &H3C4, 2
  OUT &H3C5, 2 ^ plane%
  FOR y% = 0 TO 199
    FOR x% = 0 TO 39
      GET #1, , value
      POKE y% * 80 + x%, ASC(value)
    NEXT x%
  NEXT y%
NEXT plane%

' PALETTE
OUT &H3C8, 0
FOR i% = 0 TO 47
  GET #1, , value
  OUT &H3C9, ASC(value)
NEXT i%

' WAIT
DO: LOOP WHILE INKEY$ = ""
DEF SEG
CLOSE #1
END ' OF VIEW_SCR.BAS

```

BACKDROP IMAGE SPECS:

=====

The backdrop images are essentially a "screen full of tiles". So, draw them in row/col priority, then by scanline, then by plane. (almost the exact opposite of the fullscreen images) Identify these files by filesize = 32000 bytes. Again, modified QB code:

```
' VIEW_DRP.BAS
SCREEN 12

' PALETTE
DIM palet AS STRING * 1
OPEN "gamepal.pal" FOR BINARY AS #1
OUT &H3C8, 0
FOR i% = 1 TO 48
    GET #1, , palet
    OUT &H3C9, ASC(palet)
NEXT i%
CLOSE #1

' PIXEL DATA
OPEN "drop5.mni" FOR BINARY AS #1
DIM value AS STRING * 1
DEF SEG = &HA000
FOR row% = 0 TO 24
    FOR col% = 0 TO 39
        FOR y% = 0 TO 7
            FOR plane% = 0 TO 3
                OUT &H3C4, 2
                OUT &H3C5, 2 ^ plane%
                GET #1, , value
                POKE row% * 80 * 8 + y% * 80 + col%, ASC(value)
            NEXT plane%
        NEXT y%
    NEXT col%
NEXT row%

' WAIT
DO: LOOP WHILE INKEY$ = ""
DEF SEG
CLOSE #1
END ' OF VIEW_DRP.BAS
```

ACTOR SPECS:

=====

This list is far from complete. I'm not sure if the missing values are just unused, or if they do something "invisible" like triggers or whatever, or maybe only work with certain czone's or some other limit on their use, but the values not shown don't APPEAR to do anything. Would require some thorough experimentation to figure out the rest of the values. At any rate, here's what I know so far...

OBID DESCRIPTION

==== =====

0005 *** DUKE ***

0006 *** DUKE ***
000E nuclear waste can, empty
0013 green box - rocket launcher
0014 green box - flame thrower
0016 green box - normal weapon, yeh thanks for nuthin
0017 green box - laser
001C blue box - health molecule
001F BADGUY - fast green cat, ->
0020 BADGUY - fast green cat, <-
0025 white box - circuit card
0026 BADGUY - flamethrower
0027 BADGUY - flamethrower
002A red box - bomb
002D BONUS - blue globe
002E BONUS - blue globe
002F BONUS - blue globe
0030 BONUS - blue globe
0031 BADGUY - bouncing sentry robot
0032 teleport
0033 teleport
0035 white box - rapid fire
0036 rocket launcher turret
003A BADGUY - bouncing sentry robot
003E BADGUY - bomb dropping spaceship
0040 bouncing spike ball
0042 electric current door
0043 BADGUY - green slime ball
0044 BADGUY - green slime container
004B nuclear waste can, green slime inside
004E BADGUY - snake
004F camera - on ceiling
0050 camera - on floor
0051 BADGUY - green hanging suction plant
0052 TRIGGER - causes object to right to appear only in med/hard difficulty
0053 TRIGGER - causes object to right to appear only in hard difficulty
0057 Duke's Ship
005D force shield - need cloaking device
005F rocket - falls over and explodes
0061 BADGUY - cross walker
0062 BADGUY - eyeball bomber plant
0065 BADGUY - BOSS Episode 2
0066 explosive charge
0067 explosive charge
0068 explosive charge
006A explosive charge
0072 white box - cloaking device
0073 BADGUY - sentry robot generator
0074 explosive charge
0075 pipe dripping green stuff
0077 circuit card door
0078 circuit card keyhole
0079 white box blue key
007A blue key keyhole
0080 auto-open vertical door
0081 keyhole mounting pole
0080 automatic door
0082 vertical fan
0083 swivel gun
0084 sliding floors by ladders
0085 sector marker
0086 BADGUY - skeleton
0089 explosive charge
008A explosive charge
008B exit

008C swivel gun mounting post
008E explosive charge
008F explosive charge
0090 rocket
0094 blue box - empty
0096 BADGUY - metal crunch jaws
0097 BADGUY - floating split laser ball
009A BADGUY - spider
009B blue box - N
009C blue box - U
009D blue box - K
009E blue box - E
009F BADGUY - blue guard
00A0 blue box - video game cartridge
00A1 white box - empty
00A2 green box - empty
00A3 red box - empty
00A4 blue box - empty
00A8 red box - cola
00AB BADGUY - blue guard
00AC blue box - sunglasses
00AD blue box - phone
00AE red box - 6 pack cola
00B0 BADGUY - green ugly bird
00B5 blue box - boom box
00B6 BADGUY - blue guard
00B7 blue box - TV
00B8 blue box - camera
00B9 blue box - PC
00BA blue box - CD
00BB blue box - M
00BC rotating floor spikes
00BD BADGUY - leaping gargoyle
00BE BADGUY - stone statue
00C8 BADGUY - BOSS Episode 1
00C9 red box - turkey
00CB BADGUY - bird
00D0 floating exit sign
00D1 rocket elevator
00D2 message box
00D4 lava surface
00D5 BADGUY - flying message ship "YOUR BRAIN IS OURS"
00D6 BADGUY - flying message ship "BRING BACK THE BRAIN"
00D7 BADGUY - flying message ship "LIVE FROM RIGEL IT'S SATURDAY NIGHT"
00D8 BADGUY - flying message ship "DIE"
00D9 BADGUY - blue guard
00DB smasher
00DC BADGUY - flying message ship "YOU CANNOT ESCAPE US YOU WILL GET YOUR BRAIN SUCKED"
00DD water depths
00DE lava fall
00DF lava fall
00E0 water fall
00E1 water fall
00E3 water drip
00E4 water fall spash
00E5 water fall bubble
00E6 water fall
00E7 lava riser
00E9 water surface
00EA water surface bubble
00EB green slime liquid
00EC radar antenna
00ED message box

```

00EF special hint globe
00F0 hint machine
00F1 TRIGGER - generates windblown spiders when touched
00F4 BADGUY - small unicycle
00F6 flame jet
00F7 flame jet
00F8 flame jet
00F9 flame jet
00FA ?
00FB ?
00FC floating exit sign
00FD BADGUY - caged claw monster, active
00FE flashpot
0101 water on floor
0102 water on floor
0105 caged claw monster, inactive
0106 fire on floor
0107 fire on floor
0109 BADGUY - BOSS Episode 3
010F BADGUY - small flying ship
0110 BADGUY - small flying ship
0111 BADGUY - small flying ship
0112 blue box - T shirt
0113 blue box - videocassette
0117 BADGUY - BOSS Episode 4
0128 floating arrow
012B BADGUY - swamp monster
0160 blue box - video

```

UNPACK SOURCE:

```
=====
```

This utility was originally written for Cosmo, but works just fine on DN2 and others that use the same "compact" file format. Updated slightly to compile nicely with BCB5, command line: bcc32 unpack.c Should compile clean on other "modern" compilers.

```

// unpack.c
// extracts the individual files from the compacted
// archive for Apogee games like Cosmo (and DN2, et al)
//

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    char filename[12];    // 12 bytes
    long fileoffset;      // 4 bytes
    long filesize;       // 4 bytes
} TDirectoryEntry;      // 20 bytes

#define NUM_DIRECTORY_ENTRIES 200

typedef TDirectoryEntry TDirectory[NUM_DIRECTORY_ENTRIES]; // 4000 bytes

TDirectory Directory;

void Usage(char *message);

int main(int argc, char *argv[])

```

```

{
    FILE *in, *out;
    char filename[12+1];
    long fileoffset, filesize;
    void *buffer;
    int i;

    // MAKE SURE USER SPECIFIED INPUT FILE
    //
    if (argc < 2)
    {
        Usage("No input file specified.");
        return -1;
    }

    // OPEN INPUT FILE
    //
    in = fopen(argv[1], "rb");
    if (!in)
    {
        Usage("Unable to open input file.");
        return -1;
    }

    // READ DIRECTORY
    //
    fread(&Directory, sizeof(TDirectory), 1, in);

    // PROCESS DIRECTORY
    //
    filename[12] = '\0';
    for (i=0; i<NUM_DIRECTORY_ENTRIES; i++)
    {
        fileoffset = Directory[i].fileoffset;
        filesize = Directory[i].filesize;

        // END OF DIRECTORY?
        //
        if (!fileoffset || !filesize)
            break;

        // LOCAL COPY OF FILENAME IS SAFELY ZERO TERMINATED
        //
        strncpy(filename, Directory[i].filename, 12);
        printf("Working on %s...\n", filename);

        // WRITE OUT FILE CONTENTS
        //
        out = fopen(filename, "wb");
        if (out)
        {
            buffer = malloc(filesize);
            if (buffer)
            {
                fseek(in, fileoffset, SEEK_SET);
                fread(buffer, filesize, 1, in);
                fwrite(buffer, filesize, 1, out);
                free(buffer);
            }
            fclose(out);
        }
    }
}

// ALL DONE

```

```

//
fclose(in);
return 0;
}

void Usage(char *message)
{
    printf("UNPACK - apogee game file unpacker\n");
    printf("  Usage:  unpack filename\n");
    printf("    Example:  unpack cosmo1.vol\n");
    printf("    Example:  unpack cosmo1.stn\n");
    printf("    Example:  unpack nukem2.cmp\n");
    if (message)
        printf("\nError: %s\n", message);
}

/* end of unpack.c */

```

FINAL THOUGHTS....

HOW TO FIGURE OUT THE TILE ATTRIBUTES:
 =====

Well, if ya really want to... just zero them all out and see what happens. Hint: the game will crash a lot. So you'll have to figure it out a bit at a time. I leave that joyful task to you! ;-P

You'll need this info if you want to create your own tilesets.

FWIW: The first 2000 bytes appear to be a WORD for each of the 1000 nomask tiles. The last 1600 bytes could be 5 bytes for each of the 320 masked tiles (though I can't imagine what you'd do with 40 bits of attribute flags). A common value for "solid" tiles (solid physically, as in you can't walk through them) is 0x002F, and maybe the low nibble "F" is 4 bit flags for solid on top/bottom/right/left. Personally, I'd start by trying to find the "sticky" bit for ladders and hanging poles - try to map the attributes to the tiles, note the tiles which are ladders, and compare their attributes to non-ladders, now you know what one bit does, go from there.

DN1 AND COSMO:
 =====

I had written a number of DN1 map viewer and editors including a Windows-based editor, but there's a better and more current one out there by Bryan Rodgers <rodgersb@ses.curtin.edu.au> called DN1MOD currently available at:
http://members.iinet.net.au/~markim/admiral_bob/files

So I see no purpose in trying to document the DN1 specs.

Though for anyone interested, one of my later DN1 DOS-based editors can currently be found at this link (not my site):

<http://members.fortunecity.com/stuntracer/duke1.html>

That editor requires you to have the registered version of DN1 or it won't allow saving - a reasonable concession I made to Apogee at the time to keep them from taking me to court over it!) However,

it might be of interest just for the extra levels it contains.

As far as Cosmo goes, I had a map viewer and editor for it too, but much like a lot of work I did on DN2 it has since been lost to the great void of floppy disk backup storage. Though if you get this DN2 spec, you should be able to figure out Cosmo without too much trouble since they're very similar.

None of these links are guaranteed to be functional after...

Sep 16 2002

DB